A Holistic Approach to Software Quality at Work

Manfred Broy, Florian Deißenböck, Markus Pizka
Institut für Informatik
Technische Universität München
Germany – 85748 Garching
{broy, deissenb, pizka}@in.tum.de

ABSTRACT

Software quality is a crucial issue in a society that vitally depends on software systems. Software quality definitions, standards, and metrics have contributed to the improvement of our understanding of this issue. However, there is a miss-match between the formalization of software quality issues and the practical demands of software quality. Software quality is not what we measure but what we experience when developing, operating, and using software systems over a long period of time. We argue, that it is dangerous to try to capture software quality merely by a set of numbers in terms of software metrics believing that these give an authentic picture. Though these numbers are helpful achieving truly high quality requires in addition a deep understanding of the field as well as valid knowledge on how to attain and assure software quality in development. If the top management of software dependent companies does not have an understanding of what software quality is about, all metrics of the world will not save them from their project failures.

KEYWORDS

Software Quality, Software Metrics, Software Development Processes

1 SEVEN QUALITIES – THAT'S IT?

Over the last 40 years, the role of computer programs has rapidly evolved from tools for scientific calculations into a multi-billion dollar market providing software for mission-critical processes and products to the vast majority of other industries and businesses. With the rapid growth of software in volum and criticality, and the dramatically increased life expectancy of software products, quality has become imperative.

Given its brief history compared to traditional industries, it comes as no surprise that in many companies the importance of software quality is not fully recognized and understood in spite of its vital role. Even among experts it is still not really clear what software quality actually means in all its facets and how it can be achieved. Certainly, there are numerous quality standards, e.g. ISO 9126 [1], that aim at rendering software quality more precisely. Usually these definitions list the typical seven qualities functionality, correctness, reliability, usability, efficiency, maintainability, and portability.

Though these properties a certainly desirable, their precise meaning and the requirements they impose on software products are all but clear. For example, the ISO standard refines maintainability into changeability and three further criteria. According to the standard, changeability is supposed to be measurable by regarding the *change recordability* metric, which is given by A divided by B where A is the number of functions changed having confirmed comments and B the total number of functions changed [2]. While commenting changes is useful to some extent, it is absolutely unclear how important this aspect really is for maintainability at large and what the other aspects of equal or even greater importance are. It just seems rather obvious that this single metric is neither sufficient nor necessary. In fact, its expressiveness is limited. One could blame this shortcoming to a large unspecific standard. But even well-known and broadly accepted metrics, such as McCabe's Cyclomatic Complexity [10], are neither sufficient nor necessary. Counterexamples in both directions are trivial to construct.

Of course, maintainability is not the only quality attribute in need for a sound clarification. Most of the other qualities, with the exception of the more intuitive attributes correctness and efficiency, struggle with similar troubles at the level of their definition, already.

Once a quality attribute is clearly understood the question of how to enforce it appears to be an even greater challenge.

Outline

The remainder of this paper develops a new perspective for more effective software quality management. After giving a few examples for software quality in real life software projects in section 2, we will take a more detailed look at the extensive work on software quality

that already exists and discuss possible reasons for its – at least partial – ineffectiveness in section 3. After this, section 4 depicts the three main ingredients to a holistic approach to software quality comprising a theory of software quality, sound criteria and rigid quality controlling. Section 5 illustrates this approach by means of a detailed quality model for the term maintainability. Our experiences with this model in large scale commercial projects are summarized in section 5.

2 REAL LIFE SOFTWARE QUALITY

Although it is no secret that many commercial software projects and products suffer from poor quality it cannot be overstressed how far away most real life and even large-scale software projects are from implementing acceptable quality standards.

While some spectacular software bugs, like the overflow that caused the ARIANE 5 rocket failure [17], became widely visible and lead to an increased awareness for specific techniques in certain situations (e.g. rigorous testing of critical systems and advanced verification techniques) the bulk of commercial software still seems to be built with little quality considerations in mind. We draw this conclusion from our own personal experiences as well as statistical material.

2.1 Functionality / Correctness

The Standish Group reports on software project cancellations and cost overruns in 80% of the cases are well known [3][4]. According to these empirical studies, more than 30% of the projects investigated produced software that provided at most 50% of the originally specified functionality. In addition to this, one can expect that many of the 80% cancelled and late projects had severe quality deficits, too

The issue of functionality and correctness is of course crucial. But what does complete functionality and correctness mean precisely? One could argue that all of the stated requirements (and no others) had to be implemented the way they were specified. This road leads to the techniques that our colleagues in formal methods develop; i.e. develop a precise formal specification and then do a formal, or even automated verification of the implementation against the specification. Though this sounds promising, it underestimates the most important issue: getting the right requirements and getting them right. Only after the functional requirements are appropriate and formalized verification becomes an issue. But since we do not really know how to judge whether we got "the right requirements" and due to the inherent troubles to confirm the validity of a requirements specification against the actual desires of users providing correct functionality remains a challenge.

2.2 Maintainability

Even in the rare cases that a software project could be considered successful according to these criteria, that is complete functionality in time and budget, the quality of the outcome deserves a second, separate look.

The bulk of the costs for a software system – 80% – does not go into initial development but into maintenance [6]. Because of this, the maintainability of a software system is of paramount importance to many organizations whose processes depend on software. Despite of this fact, our survey on software maintenance practices in 2003 [7] revealed that 70% of the participating software organizations did not regard the maintainability of the software they produce at all.

To us, it is still a mystery how large companies deliberately mobilise capital over and over again to replace old "legacy" systems with new ones. As a matter of fact, some of the new systems expose many of the undesirable properties of a typical "legacy" system, just after being released.

2.3 Efficiency and Performance

Efficiency is another example for frequent shortcomings though processing speed and memory consumption are rather intuitively comprehensible. At least three large scale commercial projects are known to the authors where the performance of the software is unacceptable for delivery. In all of these three cases, it is tried to solve this problem by switching to more powerful hardware; hence, without tackling the root of the actual quality shortcoming.

Again, the reason for this is a lack of understanding what software quality is, what the criteria are and how it can be influenced.

2.4 Possible Explanations

Why is it that the need for quality is widely known and accepted but quality seems to be missing in practice?

First of all, quality costs. Higher efficiency and increased security may easily multiply development costs. At the same time our software engineering discipline is still unable to answer basic economic questions such as "how much more costs 10% increased processing speed?" or "how many more bugs will be detected before shipping if we increase our testing efforts by 20%". Other industries are able to precisely explain the increased price and its corresponding benefit. For example, a 3 litre car may cost 3.000$ more than its 2 litre counterpart. You therefore receive a 20% improvement in acceleration and a 30mph increased top speed.

Since we are unable to reason about the costs and benefits of software quality in a similarly precise way it is not surprising that the average software customers is usually unwilling to accept explicit charges for quality issues. As a consequence, quality requirements remain often unspecified though the target quality profile depends on the individual needs of the users of the software system and the specification of the quality requirements was obviously part of a proper requirements engineering process.

Another major source for quality shortcomings is the simple fact that we still do not know what the right criteria for high quality software are. For example, it is accepted that GOTO is harmful [18]. But, does a comprehensive documentation really increase program comprehension? Does UML modelling contribute to better architectures? We argue that most of the rules commonly used during quality management have not been derived from a quality goal but selected for one of the following two reasons: 1) seen elsewhere … so it cannot be wrong or 2) easy to check. Consequently, many of the common rules hardly match the actual quality needs. Since most software developers are aware of the little impact of these rules, they simply ignore them.

In fact, many software organizations frankly admit that the primary purpose of their quality management efforts is to get an ISO 9000, CMM, or some other certificate that can be used as a selling point. The actual improvement of the quality of their products plays a secondary role. Though this attitude tastes bitter it is indeed comprehensible considering the debatable impact of the existing quality guidelines.

It should not be left unmentioned that there are of course many other reasons for the lack of quality in software products, such as weak qualification of development personnel, which we do not elaborate in greater detail, here.

3 WORK ON SOFTWARE QUALITY

Definitions of software quality and ways to enforce it are investigated in various research projects that deal with software metrics, quality models, quality management processes, or work that is dedicated to a certain topic such as testing or software visualization.

3.1 Software Metrics

Following Lord Kelvin's statement

"*The degree to which you can express something in numbers is the degree to which you really understand it.*"

academics and practitioners alike try to better understand software quality by means of software metrics for more than three decades [8]. Starting from the simplest (but still valuable) *Lines of Code* (LOC) metric to complex object-oriented metrics like *Coupling between Objects* (COB) [9] the field produced a plethora of process and product metrics. This includes well-known works like Halstead's *Software Science* [19] and McCabe's *Cyclomatic Complexity* [10]. Standards bodies, such as the ISO [2] and IEEE [11], devoted several standards to the field of software quality in general and metrics in particular.

Unfortunately many of the commonly used metrics suffer from at least one of the following deficiencies:

- The metric violates the most basic requirements for measures defined in measurement theory [21]. Typical examples are calculations that do not respect different levels of measurement. A drastic example is the Maintainability Index [20], which tries to define the maintainability of a system as:

  $$MI(S) = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC) + 50 * sin(sqrt(2.4 * perCM))$$

  aveV: avg. Halstead volume aveV
  aveV(g'): avg. cyclomatic complexity
  aveLOC: avg. LOC
  perCM: avg. percentage of lines of comments

- The metric has never been validated. It remains unclear if it actually measures what it is supposed to measure [12].

- The metric does not measure what was worth measuring but what is easy to measure. Many metrics focus on syntactic aspects that can be measured automatically with a tool. Unfortunately, the more important quality issues, such as the usage of appropriate data structures, are semantic properties that cannot be analyzed automatically. Successful manual techniques like reviews [13] point out that automation is desirable but not a crucial goal.

- The metric is neither a sufficient nor a necessary criterion. It can only serve as a hint. E.g. a Cyclomatic Complexity (CC) above 50 does not necessarily indicate a weak design; vice versa an ill-designed system may still have a low CC.

Nevertheless software metrics can be of high value if properly calibrated and applied in a well-controlled environment as part of a comprehensive quality model.

3.2 Quality Models

As the inadequacy of single metrics became evident many research groups developed integrated *Quality Models* to describe software qualities in a more sophisticated way by means of a set of hierarchically structured criteria and

metrics. Examples are the *Factors-Criteria-Metrics* approach [15], Boehm's quality model and models used in combination with software measurement tools such as the *SotoGraph* [16].

An example is the refinement of "supportability" into "testability", "extensibility", "adaptability", "maintainability", and further more detailed criteria [14]. The leaf criteria of this refinement process are expected to be concrete enough to be grasped with a metric. The metric values may then be propagated and aggregated from the leafs up to the root of the refinement tree or graph to determine an overall measure, e.g. for "supportability".

Quality models are trivially superior over the usage of single metrics. However, we have not seen yet a comprehensive or even commonly accepted model for software quality at large or just one of its seven major attributes. Most existing models suffer from similar two problems like single metrics do. First, it is tried to split the target quality attribute into well-known and easily measurable criteria and metrics inverting the goal of a proper refinement into a bottom-up collection of possible contributions to the attribute. As a consequence, truly significant factors are frequently ignored and the model looses its effectiveness.

Another danger lies in the fact that it is usually tried to limit the complexity of the quality model itself for the sake of its own comprehensibility. FCM's typical 3 level structure is simply inadequate. It is just unrealistic to expect that such abstract goals like "usability" could be broken down into measurable properties in only 2 steps! Unfortunately, many quality models are biased by this illusion. The outcomes of this are models that fail to provide a sound reasoning for the causal connection among the goal, criteria, and metrics.

3.3 Process Versus Product

It is a widespread belief, that the quality of a product is a function of its production process. For example, the Total Quality Management (TQM) approach and the ISO 9000 standard concentrate their improvement efforts on organizational and process issues. Capers Jones discusses in [22] amongst others the correlation between the CMM [23] level of an organization and the number of defects in their software products. It is shown that the average defect rate decreases with increasing CMM level. However, it is also shown, that a strong level 1 organization may deliver higher quality products than weak level 5 organizations.

While the development process certainly does have a strong impact on the outcome of the production process its relevance must not be overrated. The main purpose of the process is to guarantee reliable production of a system within time and budget according to its specification. The actual quality of the outcome strongly depends on 1) the ability to precisely specify the desired quality and 2) the possibilities to enforce it through skills and assurance techniques.

3.4 Further Quality Relevant Work

Besides metrics, quality models and processes, the largest part of knowledge about software quality is generated in individual works on certain aspects of software quality. Formal verification, testing, security, performance engineering, software architecture, modelling techniques are only a few examples of the research topics that contribute to our understanding of software quality.

While these fields generate valuable insights the missing integration and consolidation of the various results limits their effectiveness and slows their dissemination into practical environments.

4 A HOLISTIC APPROACH

We are aiming at a comprehensive, authentic, tractable, useful and realistic approach towards software quality keeping it simple but nevertheless taking care of all relevant factors. This is what we mean if we talk about a holistic approach.

4.1 Software Quality Theory

In software quality we have to distinguish carefully between a) general software quality goals like maintainability, etc., b) software quality numbers (metrics), and c) actions to guarantee software quality. We have to understand how these notions depend on each other and influence each other.

A holistic model of software quality needs a theory of these relationships and mutual dependencies.

A first step can be our quality diagrams given in section 5.1.

4.2 Sound Criteria

The foundations for the specification of quality requirements as well as assurance actions are the criteria that render quality more precisely. We define three requirements for effective quality criteria.

First, criteria must not be taken from a sole empirical origin but also be based on a sound theoretical foundation. Today, many criteria are derived from "best practices" in specific fields. Although this is not necessarily wrong the insufficient understanding of the characteristics of such criteria often leads to inappropriate uses and misinterpretations. Second, the interdependencies between different criteria must be understood. Our third

requirement is that a quality criterion must be assessable, otherwise it cannot become effective. Note, that assessable does not necessarily mean automatically checkable with a tool. The inability to assess the satisfaction of criteria by automated static or dynamic source code analysis does not render it irrelevant. Generally, many useful criteria need to be assessed in semi-automated (e.g. tool supported analysis) or manual (e.g. reviews) ways.

We will only be able to evaluate and manage software quality if we can find quality criteria which are fine-grained enough to be actually assessed and develop a clear understanding of their interdependencies. These criteria need to be integrated into a holistic quality model that describes all relevant quality criteria and how they depend on each other.

Of course, this model must be based on existing software engineering knowledge, but it is vital to design it in top-down manner without limiting oneself to "seen elsewhere" criteria. This is particularly important for criteria that are not assessable in an automated way.

4.3 Real-Time Quality Controlling

Even a truly complete and correct quality model will not improve software quality if not enforced. We believe such models need to be designed alongside a rigid, continuous, pro-active quality controlling process. Only if quality defects are detected in a timely manner they can be dealt with effectively.

Unfortunately there is widespread reluctance to install continuous quality controlling processes in software development as they are accepted in various other disciplines. This reluctance is usually justified by the costly nature of such processes. We believe that the inability of most project managers to foresee the long-term benefits of such measures is rooted in the remarkable insufficient understanding of software quality itself. If a holistic quality model could demonstrate the ROI for initially costly quality activities the mere change of mind of project manners would certainly improve overall software quality.

Key activity of the quality controlling process is the continuous assessment of the system. Where recent quality assurance activities are often carried out in an unstructured manner a holistic quality model would clearly define which criteria need to be assessed in what way. Depending on the criteria and availability of tools this activity consists of automated data collection and manual reviews whereas manual activities may be supported by tools (semi-automated assessment).

As the sole assessment of single criteria is not sufficient to evaluate system quality the assessment results need to be aggregated to be of real use. Since the quality model not only lists the criteria but describes their interdependencies aggregation rules can be derived from the model itself.

All data and its aggregations must be made available in a central place so the ones in charge of quality controlling are enabled to quickly determine the systems quality or its deficiencies and react in an appropriate manner.

As continuous quality controlling is a highly complex and costly activity the availability of appropriate tools is compulsory. Controlling can only be successful if it applies scalable assessment tools and an adequate environment for collecting and aggregating assessment data which is centrally available.

Another important aspect of quality control concerns a meta quality level: The quality control process must define measures to assess itself and the underlying model such that deficiencies in either part can be as quickly determined and corrected as in the system they work on.

5 MAINTENANCE QUALITY MODEL

Because such a holistic approach to software quality as a whole cannot be accomplished by a single research group our work concentrated on creating an exemplary quality model for the still complex quality aspect maintainability.

As pointed out before, we found designing the maintainability quality model (MQM) in a strict top-down manner as being crucial. We purposely ignored the metrics commonly associated with maintainability and focused on substantial quality criteria which we elicited from a broad variety of resources including well-known work in that field, our own experiences and extensive interviews with our project partners in industry.

5.1 Evolution of the MQM

For our initial design of the maintenance quality model we picked up the basic concept of FCM and other tree-like quality models. In contrast to previous quality models for software maintenance we did not choose "maintainability" as the root node of the model. We believe for "maintainability" (as for most other –abilities) it is less valuable to measure the attribute itself on some abstract scale than to determine the resources spend on the corresponding activities. Indeed interviews with our project partners in industry unveiled that project managers are not interested in "maintainability" per se; the crucial parameter for them is the "maintenance effort" (best measured in some currency).

The idea was to successively split up the quality goal "maintenance effort" in smaller, more manageable (sub-) factors. Each time a factor (or criterion) is split into sub-criteria a sound explanation for the refinement is required.

This fine-grained tree-structured model clearly points out how sub-criteria influence more general ones vice versa the factors they depend on.

Refining the criteria eventually leads to *atomic* criteria which can be assessed by some kind of checkable metrics.

The key distinction between our endeavour and previous ones is the strict top-down development process. This not only led to a quality model with more than 250 nodes (though still incomplete) but to a very new insights into software maintenance issues.

An extract of the model is given in Figure 1. Please note that the figure is highly simplified to convey the basic idea.

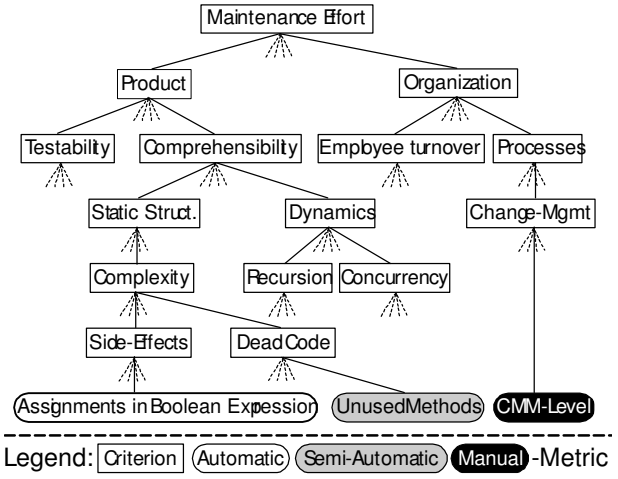

Figure 1. MQM Example

This design led to the meta-model depicted as an UML class diagram in Figure 2. It shows the basic elements of the quality model and their interconnections. The tree-structure of the quality model is described by the familiar composite pattern which has been extended by an explanation element that specifically describes how a criterion influences its super-criterion. Atomic criteria are additionally associated with metrics to assess them. The metric can be of one of the following metric classes:

- Measurement. An automated metric like LOC.
- Tool supported analysis. A manual analysis which is well supported by tools, e.g. clone detection.
- Inspection. An inherently manual analysis; i.e. part of a review.

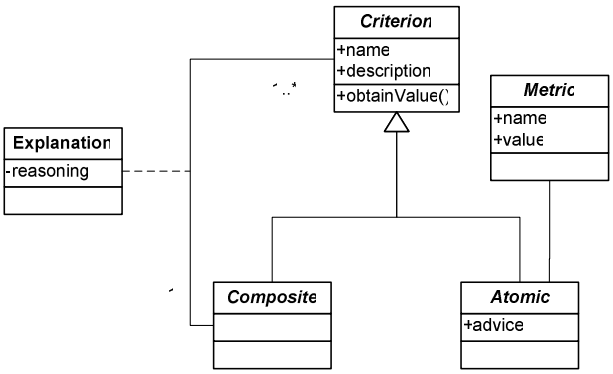

Figure 2. Initial Meta-Model

To assess the maintenance effort of a system the values determined by the metrics need to be aggregated to higher levels in the quality model. During this aggregation values need to be normalized on a common scale. Depending on the requirements this scale may be as simple as an ordinal scale (e.g. red, yellow, green) or highly sophisticated (e.g. currency units). To stay as flexible as possible the model allows the definition of the aggregation algorithm to be criterion-specific. Therefore every criterion may be equipped with a specific obtainValue()-function. This also enables users of the model to parameterize it for a particular project situation.

5.2 Current design of the MQM

During continuous extension and refinement of the MQM it became apparent that the initial, tree-like setup of the model could not be kept because it ignores interdependencies between separate sub-trees of the model. A specific example is the quality criterion which bans the usage of code constructs which rely on the evaluation order of the compiler. This hampers program comprehension and portability alike and should therefore be connected to both criteria. As loosing the tree property and switching to a general graph as basis of the model would have meant to give up on structuring and paving the ground for an incomprehensible mess itself we searched for other possibilities to organize the model by re-inspecting all quality goals collected so far. This examination revealed that the existing model was actually mixing two different kinds of nodes: maintenance activities and system properties.

We therefore split the existing model in two distinct trees; one describing the maintenance activities with “maintenance” itself as root node and one describing system properties. As we do not only consider the software system itself but also the environment (employees, infrastructure, processes, etc.) it is developed in we labelled the root node of the second tree “situation”. For both trees the design process resembles the one of the

original model. Properties and activities alike are split up to an atomic level. Atomic properties can then be associated with metrics.

The two trees are placed at the top and the left hand side of a matrix. This matrix indicates how properties influence activities (see Figure 3). Please note, that the depicted extract is strongly simplified; it does not go down to the atomic level and omits the better part of the nodes in both trees as well as the metrics and their assignment to the atomic activities.

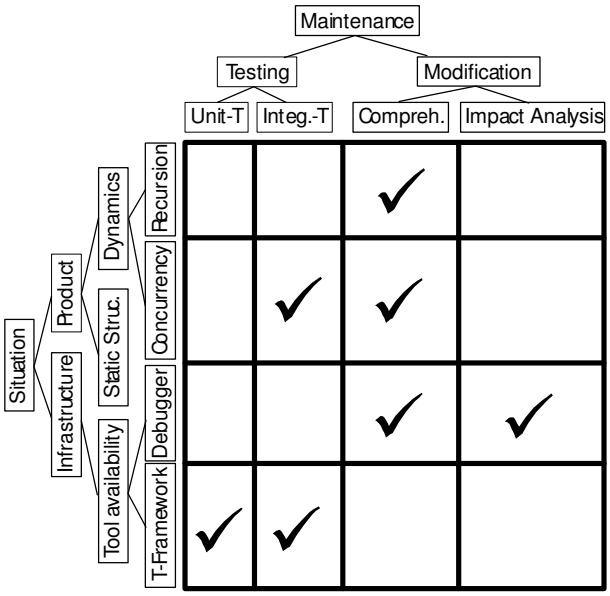

Figure 3. Example QM

As metrics are omitted the figure merely shows which properties influence which activities at all. It does not show to what degree a specific property influences the effort for a specific activity. Do determine the maintenance effort of a system (or situation) the aggregation mechanism of the original model needs to be slightly modified: Every element of the matrix is filled with the metric value determined for the atomic properties. Depending on the scales used, a general function for all elements or functions specific to each matrix element are defined to assign the values to the affected atomic activities. Further functions are used to aggregate these values in the activities tree.

5.3 MQM Conclusions

On first sight most people presented with the MQM react with the scepticism concerning the size and complexity of the model. We are well aware of this fact but claim that this complexity is inherent to the problem we tackle. It should be not surprising that a complete explanation of maintenance efforts cannot be a simple one.

We furthermore claim that the perceived complexity of the model in its entirety does not prevent it from being applied effectively. Our experiences (see below) with the model are very promising: Only because the MQM enabled us to clearly point out the broader context of specific quality criteria we were able to convince developers and project managers of their value.

But it is true that the complexity of the model demands a set of powerful tools to maintain and apply it. We therefore dedicated a lot of effort to the development of a database-backed tool that supports maintenance and extension of the MQM itself. One of its core features is an export facility that generates a hyper-linked HTML representation of the model.

Particularly important was the development of the highly extensible and configurable quality assessment tool *ConQAT*. This tool can be easily extended to integrate the plethora of available metrics and analysis tools and allows the definition of aggregation rules as defined in the model. The program is run in a non-interactive manner and produces an HTML-formatted output of all relevant data. Using it in a nightly build process enables all project participants to continuously monitor the system's state.

6 EXPERIENCES

The first experiences with the MQM were made during a six month test stage where parts of the model were tested on an information system in the field of telecommunication. From the maintenance point of view the system was an ideal candidate as it was large (3.5 MLOC[1] C++, COBOL, Java), long-lived (15 years) and had a tremendous change frequency (150 CR/year[2]).

It was interesting to see that the company had only a very limited quality assurance process that mainly focused on correctness of the software though there is highly rigid and well-found change, version and build management processes. As experienced before, the reluctance to install specific quality measures for maintainability was justified by missing resources for this kind of activities. As the managers were well aware of costly maintenance problems we perceived this as a problem rooted in the insufficient understanding of the benefits of specific measures and software quality in general.

The well-known phenomenon of elaborate coding convention documents that are never read let alone respected could also be found here. As expected the major

[1] Million lines of code

[2] Change requests per year

problems were missing justification for most of the guidelines and a literally non-existent mechanism to check for compliance with coding conventions.

A number of criteria defined in the maintenance quality model were initially perceived as rather insignificant by the project managers and developers but later on turned out to be important. For example, as part of source code readability, the MQM states precise rules for the naming of identifiers. Not only did the detailed explanations of the importance of identifier naming help to increase the level of conciousness of programmers for the importance of this aspect but we also detected some serious problems in the existing system. An example is a plethora of non-English identifiers which not only violate the companies own coding guidelines but prove to be troublesome in the context of ongoing discussions about off-shoring software development.

Another important quality criterion of particular importance in maintaining such a large system is source code redundancy (code cloning). Asked for an estimate about the percentage of cloned code in the system the project manager was already cautious but the total number of more than 2.000 code clones of at least 10 lines each did indeed surprise him. We believe we could only succeed in improving the awareness for this issue throughout the whole development team because the quality model clearly laid out what consequences cloned code has on the maintenance effort.

Our assumptions about the relevance of criteria which cannot be assessed in an automated were confirmed, too. One of the most pressing and current maintenance problem in the sample project had to do with the generation of natural language error messages. With the MQM, we could show that at the core of this problem lies an “inadequate algorithm”. An aspect included in the MQM but of course one, that can not me checked by any kind of automated source code analysis.

The application of the model of course delivered a number of further insights into maintainability. For example, the sample project suffered from incredibly long compile times that hampered productivity – an issue we never came across as a quality factor in previous work on maintenance. Closer examination unveiled that the problem was due to a misconfiguration of the compiler and a great number of superfluous include statements. We were pleased to see that our model proved to be robust enough to easily incorporate these new insight as new criteria.

Although we were highly satisfied with the results obtained by assessing the system in a quality model guided way, we feel that the substantial benefit of the quality model lies in the possibility to foster a deeper understanding of software quality among the developers. The model enabled us to successfully promote pro-active quality measures for the first time.

On the negative side, it became clear that most project participants found the two-dimensional format of the model itself complicated as they were accustomed to conventional sequential representations of quality guidelines. Here am HTML version of the model with hyper-links between criteria greatly helped to make it more accessible but we nevertheless consider generating a version of model that resembles traditional guidelines more closely.

7 CONCLUSION

What we need is a much better understanding of how quality is influenced by constructive and analytical quality options.

- How much does the increase of the testing phase influence the reliability?
- What is the most effective way to improve quality?
- What is the effect of a consequent model driven development on the quality?

At the end quality is a cost issue! Cost effective quality management is what we have to aim at. The real issue is, how much do I have to spend for which step and action during development and maintenance of a software system to achieve a certain quality level and profile.

8 REFERENCES

[1] Software engineering – *Product quality – Part 1: Quality Model*. ISO/IEC 9126-1, June 2001.

[2] Software engineering – *Product quality – Part 3: Internal metrics*. ISO/IEC 9126-*3*, July 2003.

[3] Standish Group International, Inc. *CHAOS.* 1995

[4] Standish Group International, Inc. *CHAOS: A Recipe for Success*. 1999

[5] Michael L. Brodie, Michael Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. Morgan Kaufmann, March 1995

[6] Thomas M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996

[7] Karin Katheder, *A Survey on Software Maintenance Practices*, Technische Universität München, November 2003

[8] Horst Zuse. *A Framework of Software Measurement.* Walter de Gruyter, 1998

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.,* 20(6), 1994. Halstead: Software Science

[10] Thomas J. McCabe. A complexity measure. *In ICSE '76: Proceedings of the 2nd international conference on Software engineering.* IEEE Computer Society Press, 1976

[11] *Standard for a software quality metrics methodology.* IEEE 1061, 1998

[12] Cem Kaner and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *Proceedings of the 10th International Software Metrics Symposium.* IEEE CS Press, 2004.

[13] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal.,* 15(3), 1976.

[14] R.B. Grady and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[15] J. A. McCall, P. K. Richards, and G. F. Walters. *Factors in Software Quality.* US Rome Air Development Center, Springfield AD/A-049-015/055, 1977.

[16] Walter R. Bischofberger, Jan Kühl, and Silvio Löffler. Sotograph - a pragmatic approach to source code architecture conformance checking. In *EWSA*, 2004.

[17] J. L. Lions. *ARIANE 5 – Flight 501 Failure*. European Space Agency (ESA), July, 1996.

[18] E. W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3), 1968.

[19] Halstead, Maurice H. *Elements of Software Science, Operating, and Programming Systems Series* Volume 7. New York, NY: Elsevier, 1977.

[20] Software Engineering Institute, Carnegie Mellon University. *Maintainability Index Technique for Measuring Program Maintainability*. January 2004 (http://www.sei.cmu.edu/str/descriptions/mitmpm.html)

[21] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. Softw. Eng*., 20(3):199–206, 1994.

[22] C. Jones. *Software Assessments, Benchmarks, and Best Practices.* Addison Wesley, 2000.

[23] Paulk, M.C.;Weber C.V.; Curtis, B.; Chrissis, M.B.:The capability maturity model, guidelines for improving the software process.Addison-Wesley 1995.